Fractal Xplorer: a ray-marching interactive 3D fractal generator.

Ruben Carvalho and José Rocha Faculty of Science, University of Porto

In this report we describe the implementation process of our interactive fractal generator, as well as the problems and solutions developed.

1 Introduction

Fractal Xplorer is an interactive 3D fractal viewer implemented in the C++ programming language, OpenGL API and GL Shading Language.

It takes advantage of Ray Marching in order to render the world view in real-time and represent complex fractal objects.

Our work is based on the recent state of the art for ray marching. This includes the work of Inigo Quilez [2], the Fractarium application [9] and other useful web blogs [8] and forums [1] that helped us reach this level of understanding in this topic. We also want to thank Code Parade at Youtube for introducting us to the concept of 3D ray-marched fractals [4] through his videos and his Marble Marcher game [3].

This small article describes the implementation of our project, as well as provides a couple of render shots and simple to understand tutorials for those who are interested in implementing their own engine.

This project is available on github.

2 The Rendering Model

Most of the techniques used in the past to render fractals used Ray-Tracing has their rendering model. Let's take a general view on how this method works.

2.1 Ray Tracing

Ray tracing is a render technique that tries to emulate how real light behaves. This technique consists of sending one or more light rays per pixel in the screen and calculating intersections and reflections of them with the scene objects by intersection with light source rays.

Because of the way it emulates real lighting, it provides the most realistic method of rendering as can be seen in Fig 1.

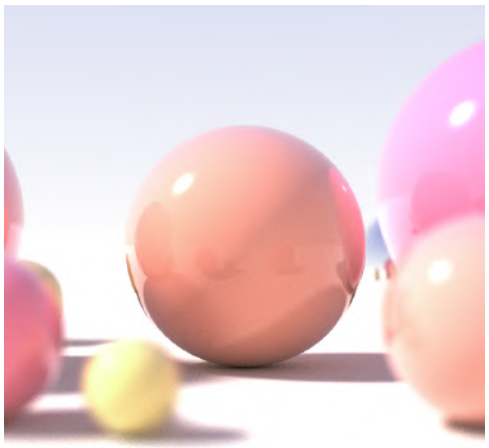

Figure 1: Ray tracing render of a couple spheres [12]

Behaviour such as reflections, refractions, scattering and dispersion are mostly *free* in terms of implementation with this technique. While this is extremely valuable when producing 3D renders for movies or still images for example, this rendering method usually comes with the letdown of an immense computation cost. As such, it is usually not used for video games for example, where real-time rendering is a must.

As shown in Fig. 2, for each pixel of the screen we need to calculate the angle of each light ray based on

the position and orientation of the scene camera and do all the intersection and reflection calculations on each of the cast rays.

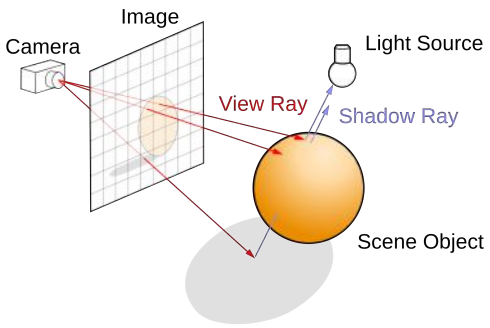

Figure 2: Ray tracing technique.

Because this is done recursively and the computer doesn't know beforehand the distance of each object in the scene to the camera, we need to keep on travelling a really small incrementation distance at each step till we intersect with a object within our scene. This adds a lot of computational load.

2.2 Ray Marching

Ray marching is a recent rendering technique that shares a lot of common algorithmic logic with the Ray tracing method. While in the ray tracing method objects are defined in a scene like a normal projection rendering, ray marching is a bit different.

In ray marching we march each of our view rays given a certain minimum distance to all the objects of the scene. Take a look into Fig. 3.

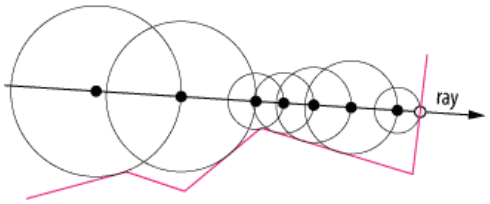

Figure 3: Ray marching technique.

Supposing we have a mathematical function that gives the minimum distance to every point in the scene, we can simply march that distance and we won't risk over-stepping any object.

This is where ray marching wins in computational simplicity: instead of doing an enormous exhaustive search little by little till we hit a wall, we already know more or less how much we can jump without hitting any wall in front of us.

Contrary to the ray tracing technique, in ray marching we define our object scene through a gathering of mathematical functions, called **SDFs (Signed Distance Functions)**, that given a point in space return the distance of that point to the object they define.

2.3 A little about fractals

Fractals are complex-space objects that are usually defined by a mathematical function. This is great news for our ray marching model! In fact, most of the ray marching done today is based on the generation of fractals, due to the enormous complexity it brings when you try to represent something that follows your common real scene.

As such, the rendering of a fractal is as easy as defining your SDF, which are mostly already defined through mathematical studies of these complex objects.

2.4 The decision

Because of the real-time rendering possibilities it allows and the easiness of defining the fractal shapes in our scene, we chose to go with the Ray Marching rendering technique.

We later found out that a lot of *free* effects are also easily achievable through this technique. These will be explained as needed throughout the development section.

3 Development

With our rendering model chosen, we now had our eyes focused on the development of our application.

3.1 Chosen tools and frameworks

For our main development language we used **C++**. Not only it provides an object oriented style of programming in order to follow good practices and easier development, but it also provides the low level manipulation necessary to work with buffers and the types of data that are fed to the graphics card API (**OpenGL**).

For the keyboard and mouse interaction and screen rendering we decided to go with the **SDL** library for C++ that facilitates the multi-platform development.

The main code of our engine is written as a fragment shader using **GLSL**. A shader is a program that is compiled to be ran by the graphics card.

For version control we used **git**.

3.2 The OpenGL base application

We started our implementation with the creation of our OpenGL application. It implements the following classes:

- **Display:** describes the dimensions and the context of the application screen.
- **Menu:** describes the main menu of the application with all of the options.
- **Camera:** describes the camera in the 3D world: orientation, position and velocity.
- **Mouse:** describes the position and offset of the mouse in the screen for intercation.
- **Mesh:** describes a simple mesh for shader rendering and implements the functions to transfer its data to the GPU.
- **Shader:** describes the vertex and fragment shaders and implements a simple communication with the graphics card to read and compile the shader programs.

This application initiates the OpenGL and SDL context and creates a simple square mesh occupying the full screen where the shader will render on.

It then compiles the written shader program files in run-time and sends them for the GPU to run.

It also reads input from the user and sends it to the shader through the use of **uniform** variables: Uniform variables are read-only and have the same value among all processed vertices. They are shared between CPU and GPU.

3.3 GLSL rendering engine

In this section we describe the implementation of our rendering engine.

3.4 Vertex Shader

Because we are not doing any vertex representation of a world, we only use this shader to pass the uniform variables to the Fragment Shader.

```
void main(){
    gl_Position = vec4(pos,1.0);
    vSystemTime = systemTime;
    vSystemResolution = systemResolution;
    vCamera_pos = camera_pos;
    vMouse_delta = mouse_delta;
    vSelectedFractal = selectedFractal;
}
```

Information such as the resolution, camera position and the selected fractal option are included in the uniform variables passed to vertex shader from the OpenGL program. The vertex shader then shares the values of those variables with the fragment shader through the use of **varying** variables.

3.5 Fragment Shader

This shader represents the implementation of our rendering engine.

All the following displayed figures were rendered using this shader.

3.5.1 Ray marching algorithm

Let's take a look at the implementation of the ray marching algorithm:

```
/*
 * Ray marching algorithm.
 * Returns aprox. distance to the scene from a certain point
 * with a certain direction.
 */
float rayMarch(vec3 from, vec3 direction) {
        float totalDistance = 0.0;
        int steps;
        for (steps=0; steps < MAX_MARCHING_STEPS; steps++) {
                vec3 p = from + totalDistance * direction;
                float distance = sceneSDF(p);
                totalDistance += distance;
                if (distance > MAX_DIST || distance < EPSILON) break;
        }
        currentSteps = steps;
        return totalDistance;
}
```

As the theory suggested, we just keep marching our ray based in increments of the minimum distance to the scene till we have a distance that is less than our (meaning a hit) or a distance larger than our maximum distance (ray going to the infinity and not hitting any object).

3.5.2 Defining a fractal SDF

Now that we had our ray marching algorithm ready, the next step was to test it. For this, we first need to define a certain object to render.

We define our first fractal SDF: the **Mandelbulb** [6].

Figure 4: Ray marched mandelbulb silhouette.

We successfully got our first render of a 3D fractal! The coloring of the pixel depends in whether the marched ray from it hit the fractal or not.

But our scene has no lighting yet. How can we define the 3D shape instead of just the silhouette of our objects?

3.5.3 Ambient Occlusion

Ambient occlusion is one of the *free effects* we can get with the ray marching technique. We use a technique called **orbit trapping**.

This technique consists of storing the minimum, maximum, mean, or any other grouping of the distance from each surface point to the X,Y,Z planes and to the camera.

We store these values in a four-dimensional vector and for now we demonstrate just the usage of the last one, the distance to the camera.

This value is then directly used as the pixel *shade* to color any pixels inside the silhouette of our fractal render.

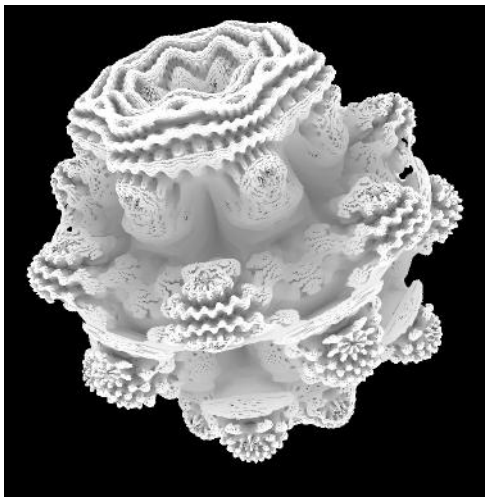

Figure 5: Orbit trap-based shaded mandelbulb.

3.5.4 Illumination

Illumination was implemented to further enhance our renderings of the fractal objects.

We implemented a simple Phong [5] illumination, that consists of varying the intensity of the color of the object based on the angle between its normal and the

light source direction. A single light source was used for performance reasons.

```
float getLight(vec3 samplePoint){
    vec3 lightPosition = vec3(0.0,10.0,0.0);
    vec3 light = normalize(lightPosition-samplePoint);
    vec3 normal = getNormal(samplePoint);

    float dif = clamp(dot(normal,light)*diffuseStrength,0.0,1.0);

    float distanceToLightSource =
        rayMarch(samplePoint+normal*EPSILON*2.0,light);

    if(distanceToLightSource < length(lightPosition-samplePoint)){
        dif *= shadowDiffuse;
    }

    return dif;
}
```

This **getLight** function implementation returns a **diffuse** value. We multiply all of our fractal pixel colors with this diffuse value to get our illumination emulation on top of the ambient occlusion.

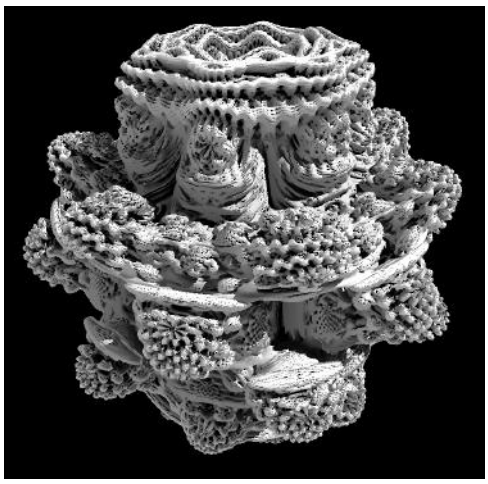

Figure 6: Mandelbulb with illumination and ambient occlusion.

3.5.5 Background color

Changing the background color is easy. We simply check if the distance marched is higher than our defined max distance. If it is, we are in the case that the ray that was sent from the pixel went to infinity. We then color all the pixels that follow this case with our background color.

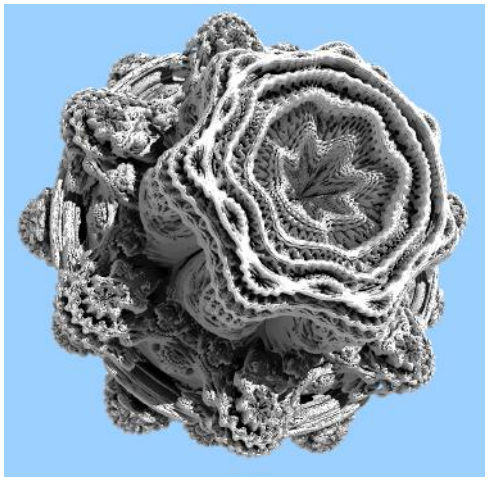

Figure 7: Added background color to our scene.

3.5.6 Coloring the fractal

Fractal renders look great with the simple illumination techniques we implemented. But to really make them stand out, we also want to color them. Because they are fractals, there are virtually no rules on how you should color them!

This process is extremely tedious to do, it requires fine tuning of certain colors to certain fractal scales, so the offset color values have to be hard-coded for the renders to look good.

As the base to color the fractals we use the previous orbit trap that calculates the minimum distance from the surface of the fractal to the X,Y and Z planes and apply offset values and interpolations with other colors on it.

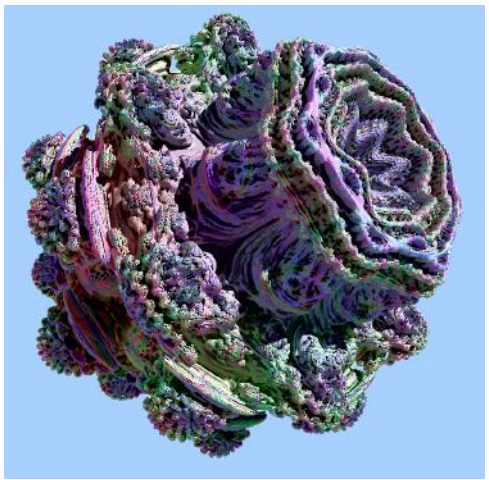

Figure 8: Colored mandelbulb.

3.5.7 Glowing

While we feel this effect is less important, we implemented it for eye-candy reasons. It also looks great with the mandelbulb fractal!

To implement the glow effect we just have to take into consideration the amount of steps marched on the rays that went to infinity. The higher the number of steps the closer they got to the fractal.

Then its as simple as interpolation a glow color value with the background color based on that number.

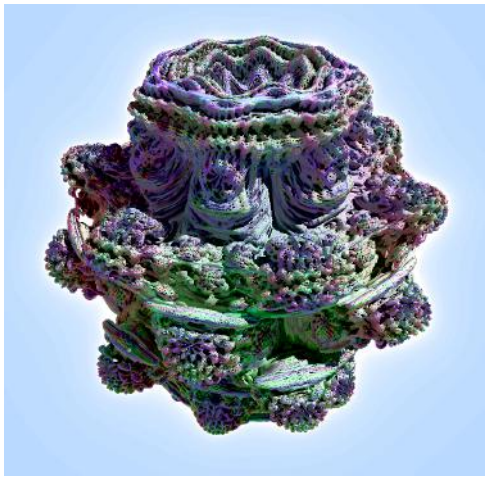

Figure 9: Glowing mandelbulb.

3.5.8 Fog

The fog was the one effect we highly underestimated. After its implementation in our engine, the sense of scale was really brought up to life.

The implementation is pretty simple, we just mix a the background color with the pixel color based on the marched distance.

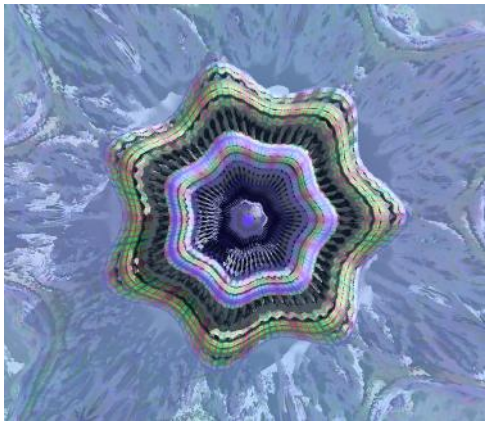

Figure 10: Fog effect sense of scale.

fractals such as the Quartenion Julia Set [7], Sierpisnki Tetrahedron [11] and the Mandelbox [10].

4 Renders

In this eye-candy section we provide renders we did using our application for display. These also include new

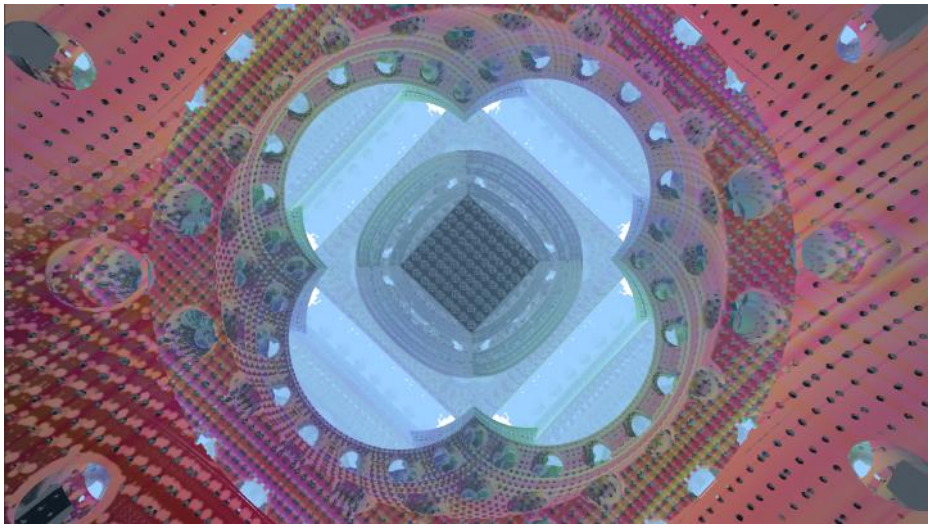

Figure 11: Rainbow and Fog Mandelbox

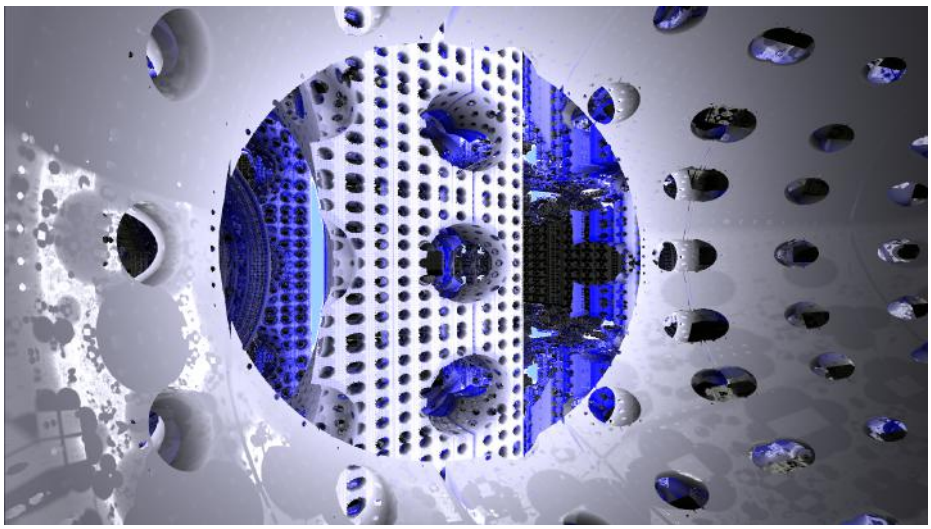

Figure 12: Looking through one of the Mandelbox holes

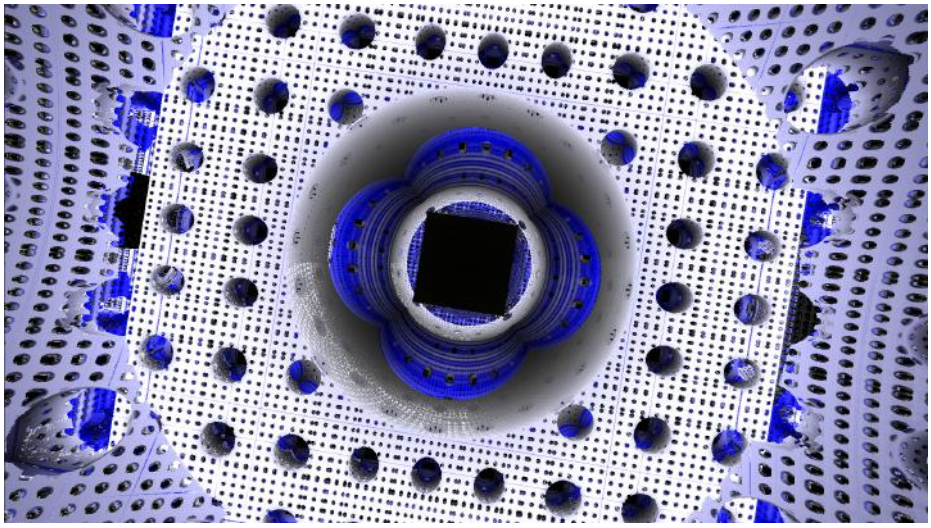

Figure 13: At the center of the Mandelbox

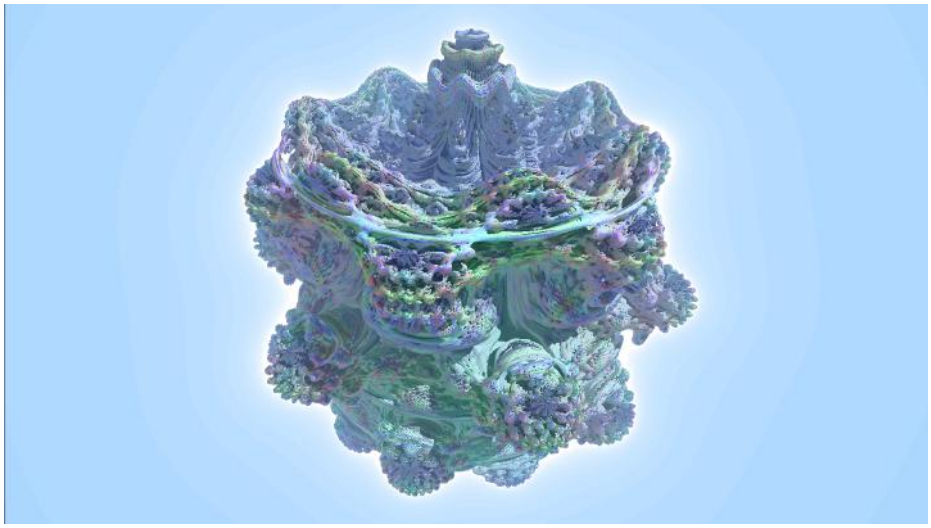

Figure 14: Mandelbulb with fog

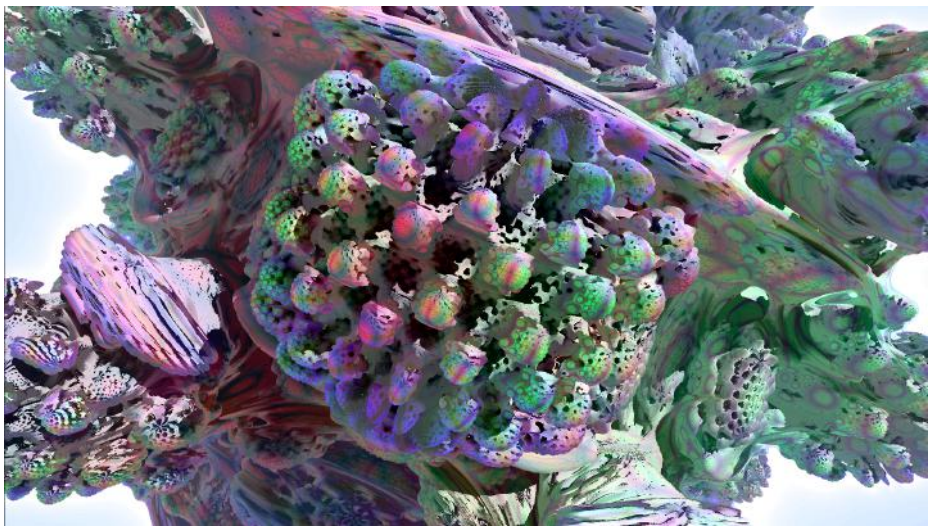

Figure 15: Mandelbulb Bulb detail

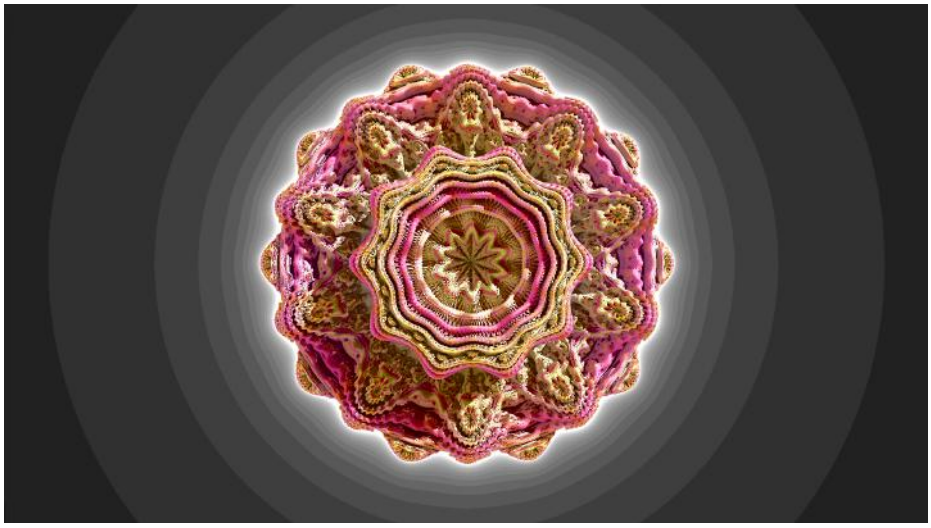

Figure 16: Power of 11 Mandelbulb

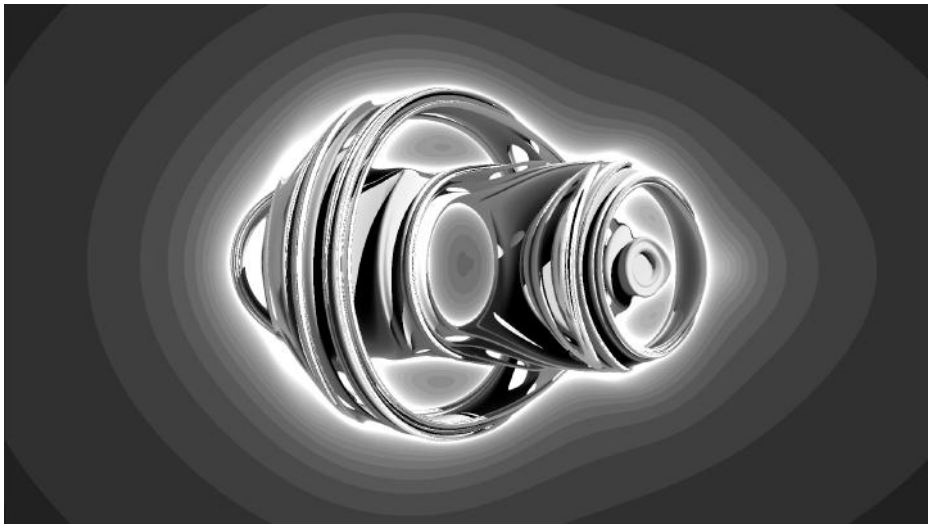

Figure 17: Julia quaternion render.

Figure 18: Julia quaternion cut at the z plane.

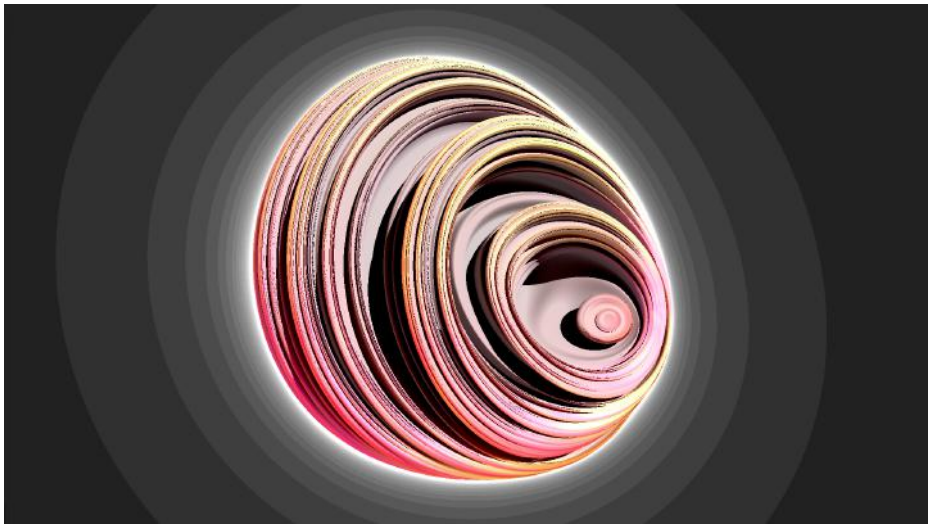

Figure 19: Some more Julia.

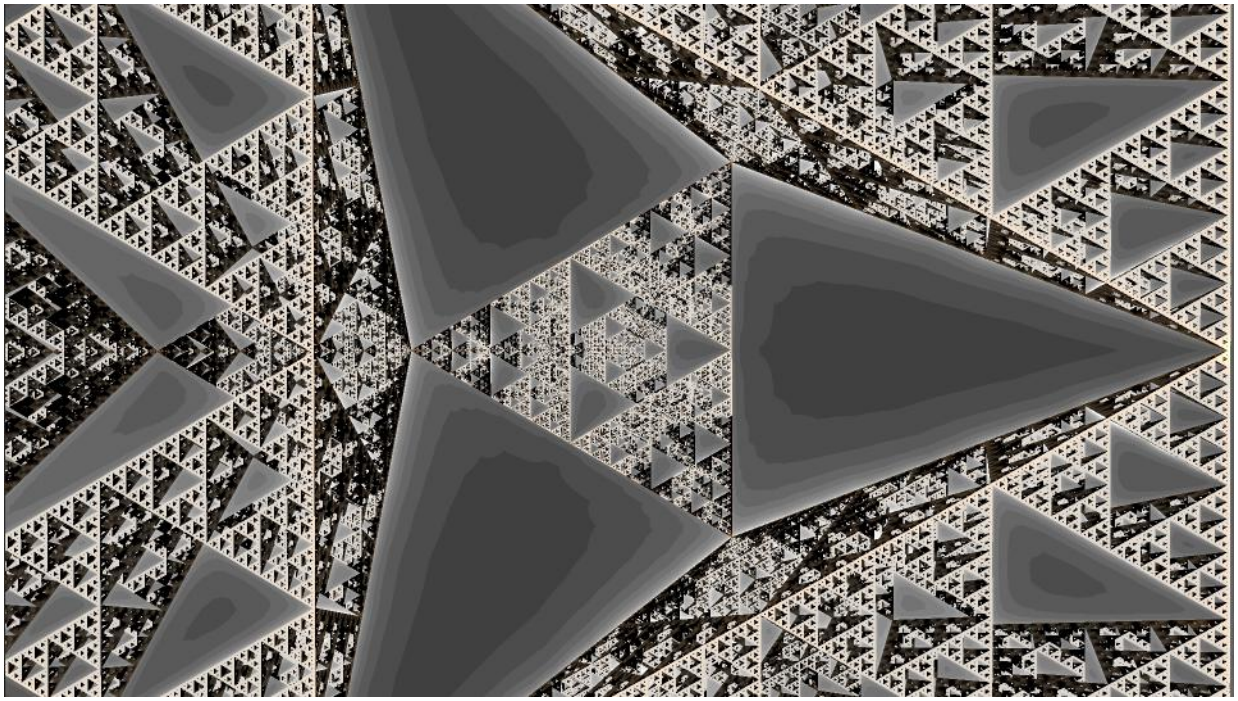

Figure 20: Diving into the Sierspinski tetrahedron.

5 Conclusions and Future work

This work was an awesome introduction to programming shaders for the GPU. We are open to more implementation and optimization suggestions, and will be maintaining this project at its github repository.

We list a few features we would like to implement in the near future.

5.1 Camera movement

The current interaction is based on rotating the fractal around with the mouse and zooming in and out with the keyboard keys.

We feel like it would benefit those who want to explore and create their own renders in our engine to implement a world camera that can be used to explore the fractal in a more controlled manner.

5.2 Multi-pass renderer

To support higher ray marching definition and resolutions, we would also like to implement a multi-pass renderer, each pass increasing the definition of the image renders.

5.3 Anti-aliasing

Anti-aliasing is a must in order to create higher quality images. It can be implemented by increasing the number of rays sent by each pixel and interpolating their results.

A small thank you

Overall, we want to thank you for showing interest in our project and reading this work report. Don't be afraid to contact us on github to expose your ideas or even fork our project.

See you in the fractal world!

References

[1] Fractal forums. http://www.fractalforums.com.

[2] Inigo quilez's webpage. https://www.iquilezles.org.

[3] CodeParade. Marble marcher. https://codeparade.itch.io/marblemarcher.

[4] CodeParade. Youtube channel. https://www.youtube.com/channel/UCrv269YwJzuZL3dH5PCgxUw.

[5] Bui Tuong Phong. Illumination for computer generated pictures. http://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf.

[6] Inigo Quilez. Mandelbulb sdf. https://www.iquilezles.org/www/articles/mandelbulb/mandelbulb.htm.

[7] Inigo Quilez. Quaternion julia set sdf. https://www.iquilezles.org/www/articles/juliasets3d/juliasets3d.htm.

[8] Syntopia. Distance estimated 3d fractals. http://blog.hvidtfeldts.net/index.php/2011/06/distance-estimated-3d-fractals-part-i.

[9] Syntopia. Fractarium. http://syntopia.github.io/Fragmentarium.

[10] Syntopia. Mandelbox sdf - fragmentarium. https://github.com/Syntopia/Fragmentarium/blob/master/Fragmentarium-Source/Examples/Historical%203D%20Fractals/Mandelbox.frag.

[11] Syntopia. Sierpisnki tetrahedron sdf. http://blog.hvidtfeldts.net/index.php/2011/08/distance-estimated-3d-fractals-iii-folding-space/.

[12] wikipedia. Ray tracing. https://en.wikipedia.org/wiki/Ray_tracing_(graphics).